Eyshila Raissa Neves Lopes
Rebeca Araújo Rodrigues Queiroz
ACH2003 - Computação Orientada a Objetos
30 de setembro de 2025

Relatório - Aplicação de princípios de POO a um código procedural

Neste projeto, foi realizada uma reestruturação completa do código original com foco na aplicação dos princípios da programação orientada a objetos. O código fornecido inicialmente era funcional, mas apresentava uma estrutura procedural, com pouca organização, código repetido e responsabilidades misturadas. A proposta da reformulação foi justamente corrigir esses problemas e transformar o código. Além da refatoração do código foi implementado Power-ups, Fases e Chefes.

01. Críticas ao código inicial do jogo

a. Tudo está concentrado em uma única classe (Main).
b. Todos os dados (como estado de projéteis e inimigos) são manipulados diretamente em arrays.
c. Os inimigos, projéteis e o jogador poderiam compartilhar uma estrutura comum como uma superclasse ou interface.
d. Constantes como INACTIVE, ACTIVE, EXPLODING poderiam estar em enums ou classes específicas.
e. Arrays de tamanho fixo podem limitar o jogo
f. Muita repetição no tratamento de diferentes tipos de inimigos/projéteis
g. Poderia ter Detecção de Colisão Mais Clara

02. Descrição da Nova Arquitetura

Descrição e Justificativa da Nova Estrutura de Classes/Interfaces

Foram criadas classes específicas para cada tipo de entidade do jogo, como Player, Projectile, Enemy, Enemy Type1, Enemy Type 2 e Power Up. Além disso, foi introduzida uma classe abstrata Enemy, que define o comportamento comum a todos os inimigos (como movimentação, explosão, desenho na tela e estado de atividade), e que é estendida por diferentes tipos de inimigos com comportamentos específicos.

A nova estrutura divide o jogo em classes especializadas, cada uma responsável por uma parte específica da lógica:

- Game: Controla o loop principal, gerencia entidades (jogador, inimigos, projéteis) e coordena atualizações e renderização.
- Player: Representa o jogador, tratando movimentação, disparos e estados (ativo/explodindo).
- Projectile: Gerencia projétil do jogador (movimento, colisão e renderização).
- Enemy Projectile: Similar a Projectile, mas para projéteis inimigos.
- Enemy (classe abstrata): Define comportamentos comuns a todos os inimigos.
- Enemy Type1 e Enemy Type 2: Implementam tipos específicos de inimigos com comportamentos distintos.
- Cada classe (Player, Projectile, Enemy) possui seu próprio método Collides With.

Uso de Coleções Java no Lugar de Arrays

A versão original do projeto utilizava arrays fixos para representar elementos do jogo, como inimigos, projéteis e efeitos visuais. Essa abordagem apresenta diversas limitações, como a necessidade de gerenciar manualmente os índices, controle de ativação/inativação e reaproveitamento de posições no array.

Na refatoração, os arrays foram substituídos por coleções dinâmicas da biblioteca Java Collections Framework, em especial a interface List e a classe ArrayList

Agora temos tamanho dinâmico (adiciona ou remove elementos com .add() e .remove(), sem se preocupar com o tamanho máximo) e Flexibilidade e polimorfismo: ao utilizar uma lista de objetos do tipo Enemy, é possível armazenar diferentes subclasses (Enemy Type1, Enemy Type

2, etc.) e tratá-las de forma unificada com o código mais limpo (elimina a necessidade de checar índices ou fazer reaproveitamento manual de posições).

03. Funcionalidades Extras no Jogo: Sistema de Power-ups

Os power-ups foram integrados ao mecanismo de configuração das fases do jogo. Eles são declarados nos arquivos de fase usando a palavra-chave POWER UP.

I. Arquitetura Base dos Power-ups

Os power-ups compartilham uma estrutura comum implementada via herança:

II. Define os atributos Básicos:

- Posição (x, y)
- Velocidade de descida
- Raio de colisão
- Estado de atividade

III. Define métodos genéricos:

- update(): atualiza a posição e verifica colisão.
- draw(Graphics2D g): renderiza o item na tela.
- CheckCollision(Player player): verifica se o jogador coletou o item.

IV. Define o método abstrato:

- Apply Effect(Player player): aplicado quando o jogador coleta o power-up.

Power-ups Implementados

1. Aumento de Velocidade (Velocity PowerUp)

Descrição: Ao coletar esse power-up, a velocidade de movimentação da nave do jogador é aumentada temporariamente.

Objetivo: Proporcionar maior mobilidade ao jogador, facilitando desvios de projéteis e movimentações rápidas entre os inimigos.

- A subclasse Velocity PowerUp sobrescreve o método apply Effect(Player) para alterar a velocidade do jogador por um período limitado.
- Um temporizador ou flag é utilizado para controlar a duração do efeito.

2. Disparo Múltiplo (Multiple Projectiles PowerUp)

Descrição: Ao coletar este item, o jogador passa a disparar múltiplos projéteis simultaneamente.

Objetivo: Aumentar o poder ofensivo e facilitar o combate contra múltiplos inimigos.

Efeito no jogo: Durante o tempo de efeito, a nave dispara três projéteis: um central e dois inclinados (em forma de leque). A subclasse Multiple Projectiles PowerUp ativa uma flag no jogador indicando o modo "tiro triplo", que é tratado durante a rotina de disparo.

04. Funcionalidades Extras no Jogo: Sistema de Fases

O sistema de fases do jogo foi projetado com foco na modularidade e facilidade de expansão. Cada fase é definida externamente por meio de arquivos .txt que descrevem os eventos da fase, como surgimento de inimigos. A lógica de carregamento e execução das fases é feita por meio das classes Config, Fase, EventoFaseFactory e Boss.

A. Definição de Configurações Gerais (Arquivo config.txt)

O arquivo config.txt é usado para definir parâmetros globais do jogo, como:

- Tempo entre atualizações dos eventos da fase: 100
- Número total de fases: 2
- Arquivos de definição das fases: fase1.txt, fase 2.txt
- Essas configurações são lidas e armazenadas na classe Config.

B. Arquivos de Definição das Fases (fase1.txt e fase2.txt)

Cada fase é descrita em um arquivo de texto separado. As linhas representam eventos programados com a estrutura: **INIMIGO <tipo> <tempo> <posição x> <velocidade _y>**

EXEMPLO

```
INIMIGO 2 1000 240 -10
```

Isso representa um inimigo do tipo 2 que surgirá 1 segundo após o início da fase, na posição x=240, movendo-se com velocidade -10 no eixo y.

C. Classe Fase.java

A classe Fase é responsável por:

- Ler os eventos da fase a partir dos arquivos.
- Armazenar os eventos como objetos.
- Controlar o tempo da fase e disparar os eventos no tempo apropriado.
- Utilizar a EventoFaseFactory para instanciar os objetos (como inimigos) no momento certo.

Essa classe é fundamental para o controle de fluxo da fase.

D. Classe EventoFaseFactory.java

Essa classe funciona como uma fábrica de eventos. Sua função é criar dinamicamente objetos do jogo com base nas linhas do arquivo da fase. Por exemplo, ao ler INIMIGO 1 33000 140 -10, a fábrica sabe criar um inimigo do tipo 1 com as propriedades corretas.

E. Classe Boss.java

O sistema de chefes foi desenvolvido com base em uma classe abstrata chamada Boss, que serve como modelo para todos os chefes do jogo. Essa estrutura orientada a objetos proporciona a reutilização de código, organização e expansão facilitada para diferentes tipos de comportamento e aparência dos chefes.

A classe Boss define atributos comuns a todos os chefes, como:

- life: quantidade de vida;

- x, y: posição atual na tela;
- radius: raio de colisão;
- active, exploding: estado do chefe;
- explosionStart, explosionEnd: controle da animação de explosão

E métodos fundamentais como:

- update(Game game): lógica de movimentação, ataque e condições da batalha;
- damage(long currentTime): aplica dano ao chefe e inicia a explosão, caso a vida chegue a zero;
- draw(): desenha o chefe ou a animação de explosão;
- drawBoss(): método abstrato implementado pelas subclasses, responsável pela aparência específica do chefe.

Disparo de Projéteis: Classe BossProjectile

Os chefes podem disparar projéteis próprios, implementados pela classe BossProjectile. Essa classe representa projéteis vermelhos lançados contra o jogador e contém os seguintes elementos:

Atributos:

- x, y: posição;
- vx, vy: velocidade em cada eixo;
- radius: raio de colisão;
- Activo: status de atividade.

Métodos:

- update(long delta): atualiza a posição e desativa o projétil se ele sair da tela;
- draw(): desenha o projétil na tela com um círculo vermelho;
- deactivate(): permite remover o projétil após colisão.

Esse encapsulamento permite que os projéteis sejam reutilizados e gerenciados separadamente da lógica do chefe.

BossType1 : Chefe da Fase

BossType1 é uma subclasse concreta de Boss usada na primeira fase. Seu comportamento é:

- Movimento: curvilíneo, baseado em um ângulo que evolui no tempo, controlado por Math.cos e Math.sin.
- Disparo: projéteis direcionados ao jogador, com intervalo entre 200ms e 700ms.
- Final da fase: quando a explosão termina, o jogo define game.running = false e a fase é encerrada.
- Visual: desenhado como um círculo ciano.

O uso da herança permite que BossType1 herde toda a lógica padrão e foque apenas no comportamento único desejado. Boss Type2 também estende a classe Boss e possui o mesmo comportamento que Boss Type1.

5. Conclusão

Benefícios da Programação Orientada a Objetos (POO) na Implementação dos Power Up

A utilização da Programação Orientada a Objetos foi essencial para o sucesso da implementação do sistema de power-ups, garantindo modularidade, reutilização de código e facilidade de manutenção. A seguir, destacam-se os principais benefícios:

- **Herança e Reutilização de Código**

A criação de uma classe base para power-ups permite que os atributos e comportamentos comuns (como posição, velocidade, colisão e renderização) sejam implementados uma única vez.

As subclasses (VelocityPowerUp, MultipleProjectilesPowerUp, etc.) herdam essa estrutura e sobrescrevem apenas o método applyEffect(), reduzindo duplicações e facilitando a criação de novos power-ups.

- **Polimorfismo**

Graças ao polimorfismo, é possível tratar todos os power-ups de forma genérica, mesmo que eles tenham comportamentos diferentes. Por exemplo, uma lista de objetos do tipo PowerUp pode conter instâncias de diferentes subclasses, e ao chamar applyEffect(player), o comportamento

específico de cada tipo será executado automaticamente.
Isso torna o código mais limpo e extensível.

- **Encapsulamento**

Cada power-up encapsula seus próprios dados e lógica, como tempo de duração, efeito no jogador e aparência. Isso facilita o isolamento de erros, testes unitários e manutenção do código, já que alterações em um tipo de power-up não afetam os demais.

Organização do Código

A separação clara entre as diferentes responsabilidades (detecção de colisão, desenho na tela, aplicação de efeito, controle do jogador) torna o sistema **mais legível e organizado**, o que é especialmente importante em jogos com múltiplos elementos simultâneos.

Benefícios da Programação Orientada a Objetos (POO) na Implementação das Fases

A adoção da programação orientada a objetos foi essencial para tornar o sistema de fases modular, organizado e fácil de expandir. Entre os principais benefícios observados:

Encapsulamento

Cada classe (como Fase, Config, Boss, etc.) tem responsabilidades bem definidas, isolando seus dados e comportamentos. Isso facilita a manutenção e evita efeitos colaterais indesejados entre partes diferentes do código.

- **Reutilização de código**:

A classe EventoFaseFactory centraliza a criação de elementos como inimigos e bosses, evitando duplicação de lógica e tornando o sistema mais escalável.

- **Facilidade de expansão:**

Novos tipos de inimigos, eventos ou fases podem ser adicionados com pouco impacto no restante do sistema. Basta criar novas classes ou adicionar novos arquivos de fase.

- **Polimorfismo e abstração:**

Embora o projeto atual esteja mais voltado à leitura direta de arquivos, a estrutura permite que diferentes tipos de eventos (inimigos, obstáculos, power-ups, chefes) sejam tratados de forma

genérica, permitindo que uma lista de eventos seja percorrida e executada sem se preocupar com o tipo exato de cada um.

- **Organização e legibilidade:**

A separação clara entre dados (config.txt, fase1.txt, fase2.txt), lógica de leitura (Config, Fase) e instanciamento (EventoFaseFactory) torna o código mais compreensível e manutenível.

- **Herança e Reutilização**

A classe abstrata Boss centraliza o que é comum, como vida, explosão e desenho genérico. Isso evita duplicações e permite que cada novo tipo de chefe herde esse comportamento.

- **Polimorfismo**

O jogo pode tratar qualquer chefe como um Boss, mesmo que a instância seja de Boss Type1, Boss Type2, ou outro. Isso simplifica o gerenciamento no código da fase.

- **Encapsulamento**

Cada chefe controla seu próprio estado (vida, posição, se está ativo ou explodindo), mantendo as responsabilidades bem organizadas.

- **Expansibilidade**

Novos tipos de chefes podem ser adicionados com facilidade, bastando estender a classe Boss e implementar update( ) e draw Boss().

De forma geral, a escolha dessa programação orientada a objetos resultou em um projeto mais estruturado, que pode ser modificado com o tempo, baixo acoplamento entre os módulos e fácil manutenção. O jogo cresceu de forma consistente, com as novas fases, inimigos, power-ups e chefes sendo adicionados sem comprometer o funcionamento do sistema existente.